KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 9934

REUSING BUSINESS MODELS

by
M. SNOECK
G. POELS
G. DEDENE

D/1999/2376/34

Reusing Business Models

Monique Snoeck, Geert Poels, Guido Dedene

Abstract

The focus of this paper is on the reuse of business models. It investigates how business models can be reused, how such reuse can be measured and what the consequences are for software development.

1. Business modelling

The main objective of any business model is to be a vehicle for communication, facilitating the mutual perception and understanding of some aspects of the business reality. As stated by Nellborn [7], communication between system developers and business people is often equivalent to the Berlin Wall: both living and working in the same place, but each having little understanding about the other's work. Communication happens by "throwing things over the wall", such as specifications, class diagrams, QA-documents and so on. Business modelling is a way to destroy this wall and improve the communication between business people and software engineers. Business professionals also benefit from the use of business models as a tool to establish definitions of important concepts clear-cut enough to allow efficient and unambiguous internal communication. Also, having a model of the business, one can start evaluating the way the business is organised and consider possible changes.

In this paper we will consider the possibilities of reuse of business models. Although models are the particular representation of one or more aspects of a specific business, reuse of existing models can be interesting. First, it is more efficient to reuse an existing model than to start from scratch. Second, reusing a model can trigger critical thinking about the own business: what makes our business similar to

another business, what makes up the difference? Finally, if software components exist for the reused business model, it is likely that (part of) these components can be reused for the own business.

In section 2 we will present two examples of business models and show how they can be generalised into a generic model for a particular type of business. Section 3 will then elaborate on the notion of generic models and argue how such models can be useful for reasoning about the way of doing business in general. Section 4 will then present some metrics to evaluate reuse in a more formal way. In section 5 we will add en extra level of detail to the generic model and investigate the effect of this on reuse. Section 6 presents corresponding refined measurement techniques. Finally, section 7 will evaluate the resulting reuse possibilities for software developers and presents some conclusions.

2. A Library = A Hotel?

In this section we present a business model for a library and a business model for a hotel administration. These business models can also be called domain models since they do not take account for the particular characteristics of a specific library or hotel business. We have chosen to represent the structural aspect of the business model by means of a class diagram using the UML notation. In a later section, the dynamic aspect of the business will be modelled by listing relevant business events and by specifying which business classes are affected by these events. Other aspects such as workflow models and/or business process models are beyond the scope of this paper.

a. The library

In the library we have a catalogue with titles and for each title the library has one or more copies. People can register to the library and become members. Members can borrow and return copies. Loans can be renewed. If a book is not on shelf, a reservation can be made for that title: the first copy that is returned to the library will then be put aside. The structural aspects of this little domain description are represented in Fig. 1.

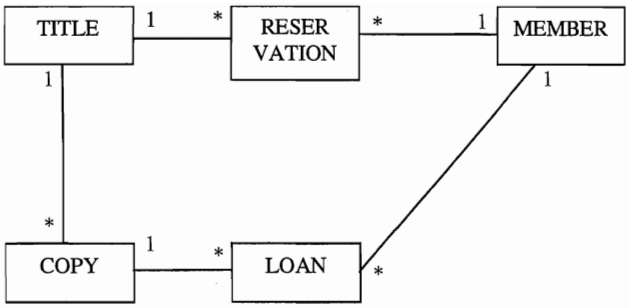

Fig. 1. A simple Library Domain Model

b. The hotel administration

A hotel offers a set of rooms that are categorised into room types. Customers make reservations for a particular room type. When the reservation is confirmed, a specific room is assigned for the customer's later stay. This situation is represented in Fig. 2.

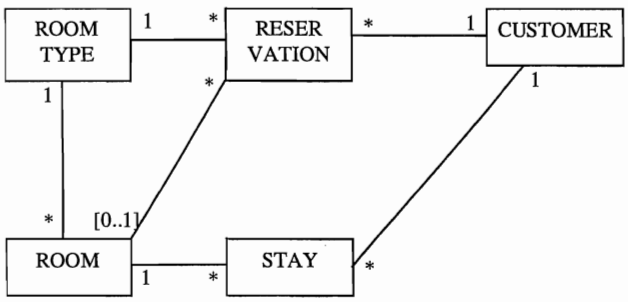

Fig. 2. A simple Hotel Administration Domain Model

c. The generic reusable model

As one can immediately notice, the class diagrams for the library and for the hotel show a very similar structure. In both businesses products are categorised to product types. Customers can "buy", in this case "use" a product. Prior to this usage there may or may not be an "order" or reservation for the product's type. The generic model is shown in Fig. 3. In this model, the association between USAGE_INTENTION

and PRODUCT represents the allocation of products to reservations or orders. The association between USAGE_INTENTION and USAGE allows to track how many of the effective usages are the consequences of a prior usage intention. For example, in a hotel it can be interesting to know for how many stays there was a prior reservation.

Such a model can immediately be transposed to other situations such as a car rental company (Fig. 4.)

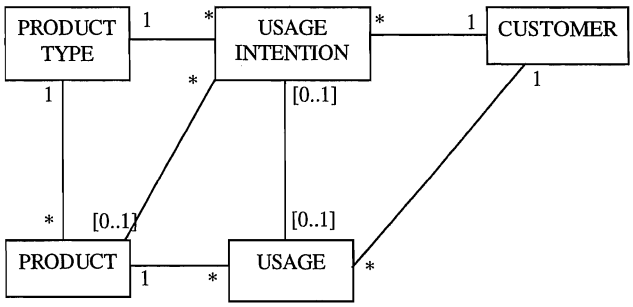

Fig. 3. A generic and reusable Model

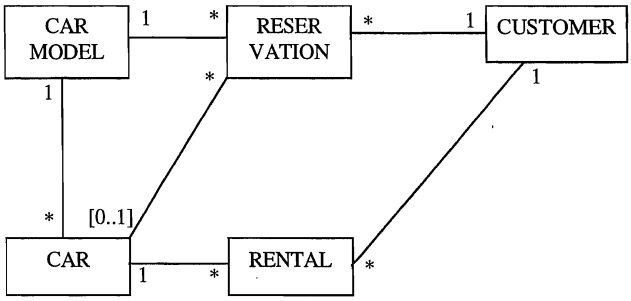

Fig. 4. A simple Car Rental Domain Model

3. Reusing Generic Models

Although the generic model can be reused in many situations, each domain will have its own particularities that must be taken care of. Tailoring the generic structural model to the particularities of the own domain can be done by adding or dropping classes and/or associations, and by considering additional business rules. For example, in the library we will probably not be interested in keeping track of how many loans are the consequences of a reservation. As a result, the association between the reservation class and the loan class has not been retained. In the cases of the hotel administration and the car rental company, the decision whether or not to retain this association depends on the information needs of the specific company.

In addition, reusing a structural model does not necessarily imply that other aspects of the business can be reused as well. For example, the way products are allocated to an intended transaction is similar in the hotel and car rental business, but very different from the library business. Both in the car rental and the hotel business it is a good practice to confirm the reservation and ensure that the requested product is available on the requested date. In a library however, such confirmation is not required: the member will simply receive the first copy that is returned and no firm assurance can be given on the date a copy will be available.

The population of classes can also be very different: a library will have many titles and most of the time only one copy per title. In a hotel and a car renting business, the ratio product type/product is much different. And whereas a library will try to maximise the number of titles available, the other two business will rather try to maximise the number of usages per product. Business goals can also be very different.

The generic model can also be extended to support multiple branches of one business. In the model of Fig. 5. we assume that product types are company-wide. However, the characteristics of a product type can be different from branch to branch: a double room in Paris will have another (higher) price than a double room in Nantes. This requires the introduction of the class PRODUCT_TYPE_IN_BRANCH. Individual products are the materialisation of such a PRODUCT_ TYPE_IN_BRANCH and are as such located in one branch.

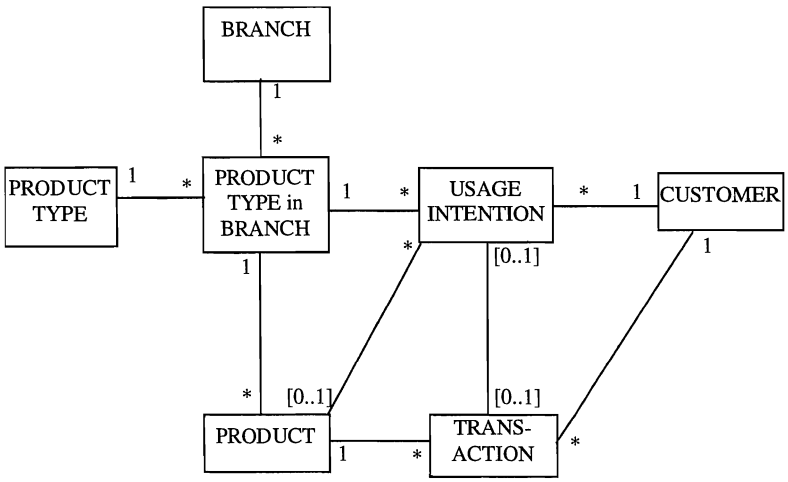

Fig. 5. Extended generic and reusable model

This model can easily be reused for the hotel administration and for the car rental company. For the car rental example it would also make sense to add an association between BRANCH and CAR that records the current location of a car. This would allow customers to return the car to another branch than where it was rented. For example, it would allow customers to rent a car in the Brussels office and return it in the Paris office.

For the library, the concept of PRODUCT_TYPE_IN _BRANCH makes less sense. It is sufficient to keep track of the location of each copy by directly linking COPY to LIBRARY (the branch). The model is adapted as in Fig. 6.

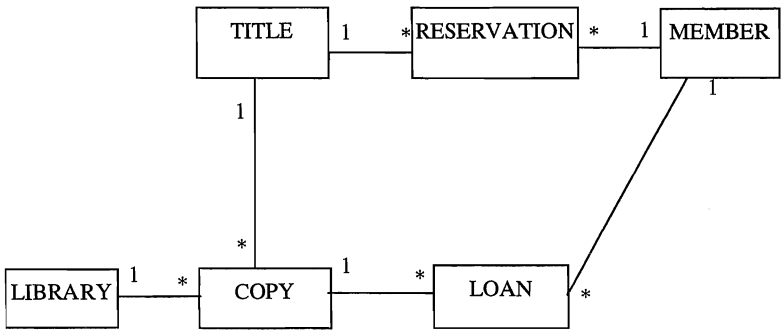

Fig. 6. Extended Library Domain Model

4. Measuring Reuse

Knowing that reuse from a generic model is possible is one thing. We might also be interested in knowing how much reuse is possible. Measuring the degree of reuse in a business model allows business professionals to quantify the similarity between their business and the business or business domain whose model has been reused. This might be interesting for a variety of reasons. For instance, managers might wish to determine other businesses that are suitable candidates for providing benchmarking data. Reuse measures also allow evaluating how close the own business organisation and processes resemble those of the market leader or the other competitors in a domain. Quantitative reuse data also helps selecting the generic model that matches ones own business most closely.

A further advantage of reuse figures is that they help software managers and engineers to assess the impact of business model reuse on the development process. First, there is the direct effect on the cost and duration of the business modelling activity. Second, there might be an indirect impact on other development activities. If reuse from a domain model is possible, then we might perhaps be able to reuse class definitions from a component library for the domain. If such a library is available, then a measure of business model reuse might provide an early estimate of the degree of code reuse in the system and consequently helps budgeting the development project.

Conceptually, there is not much difference in measuring business model reuse and the reuse of other types of software or specification artifact. The degree or level of public reuse is commonly defined as the "the sum of the sizes of the reused components divided by the sum of the size of both reused and newly built components" [12]. The general form of a public reuse measure is [8, 4]:

$$public\_reuse\_client(P) = \frac{size(P_{ext})}{size(P_{ext}) + size(P_{new})},$$

where P_{new} and P_{ext} denote respectively the newly developed and externally built parts of a software or specification artifact P. Note that reuse is evaluated here from the perspective of the 'client'.

This general form is easily instantiated for measuring the degree of reuse in a

business model. In the structural model two types of components are distinguished: classes and associations. As the structural model does not offer information regarding the sizes of individual classes and associations, each reuse of a class (or association) contributes one instance of reuse to $size(P_{ext})$ and each newly added class (or association) contributes one instance of non-reuse to $size(P_{new})$.

The reuse figures for the (simple) library, the hotel administration and the car rental company are impressive. The degree of reuse from the generic model of Fig. 3. is in all cases 100 %, both for classes and associations. It must be noted however that the measures do not capture the removal of components when instantiating a generic model or reusing another business or domain model. Probably, the non-retention of a component is less costly than the addition of a new component. However, we cannot exclude that some removal costs are involved. Therefore, we use an additional measure that measures the degree of reuse from the perspective of the 'server':

$$public\_reuse\_server(P) = \frac{size(Pext)}{size(Pext) + size(Pdel)},$$

where P_{del} and P_{ext} denote respectively the parts of a software or specification artifact P that have been removed and reused within a new artifact. Using this type of measure it can be seen that for the library example the reuse from the generic model of Fig. 3. is non-verbatim. Only 5/7 or 71 % of the associations have been reused. Similarly, the degree of reuse in the extended library model (Fig. 6) is not perfect either. If the extended generic model of Fig. 5 is used as the reuse source, then the degree of reuse is 6/6 or 100 % of classes and 5/6 or 83 % of associations (the association between LIBRARY and COPY is new) when evaluated from the 'client' perspective. From the perspective of the 'server', the degree of reuse is 6/7 or 86 % of classes (the class PRODUCT TYPE IN BRANCH has been dropped) and 5/8 or 62 % of associations (the associations between PRODUCT_TYPE and PRODUCT_TYPE_IN_BRANCH, between BRANCH and PRODUCT_TYPE_IN_BRANCH, and between USAGE_ INTENTION and USAGE have been removed).

The high reuse percentages in the examples must be interpreted with care. On the one hand, they indicate potential cost savings due to reuse. Moreover, they provide quantitative evidence of structural similarities between businesses. On the other hand, the verbatim reuse of the generic structural model, as in the (simple) car rental and

hotel administration examples, does not imply that the other aspects of the business can also be reused as such. Besides, cost savings due to reuse must be balanced against the cost of selecting the right reuse source, tailoring the source model, etc.

5. Adding an extra level of detail

In this section we will introduce an extra level of detail to the conceptual model, by modelling part of the dynamic aspects of the domain. We will identify relevant business event types and indicate which classes are affected by the occurrence of the identified business events. We record this in an object-event table (OET). This is a tabular representation with one row per business event and one column per class. Each cell of the table indicates whether or not a class is affected by the occurrence of a business event. No involvement is marked by a blank entry. If a business event creates (respectively modifies or ends) occurrences of the class, the entry is marked with a 'C' (respectively an 'M' or 'E').

In addition we will allow classes to impose sequence constraints on the business events. In the library for example, a copy should be returned before it can be borrowed. With each class we will thus associate a lifecycle expression. The default lifecycle is that objects are first created (a choice between the C-entries), then modified an arbitrary number of times (an iteration of a choice between the M-entries) and finally come to an end (choice between the E-entries).

We will first elaborate the OET and the lifecycles for the generic model of Fig. 5. Then we will evaluate the possibilities for reuse for the car rental and the library domain model.

a. The generic model

For the generic model we identify the following business event types:
create_customer, modify_customer, end_customer, create_branch, modify_branch, end_branch, create_product_type, modify_product_type, end_product_type, allocate_product_type_to_branch, modify_product_type_in_branch, end_product_type_in_branch, create_product, modify_product, end_product,

cr_usage_intention, allocate_product, confirm_availability, cancel_usage_ intention, start_usage, normal_return, abnormal_return, modify_conditions, invoice_usage, receive_payment, end_usage

The OET is represented in Table 1. The event participations marked in this OET are a minimal set of entries. If for example, we wish to keep track of how many product types are offered in a branch, it makes sense to mark the entries BRANCH/*allocate_ product_type_to_branch* and BRANCH/*end_product_type_in_ branch.* Similarly, if within the class customer we wish to keep track of the total amount of payments made by this customer (e.g. to identify "golden" customers, or to specify some discounting rules), we need to mark the entry CUSTOMER/ *receive_payment.* Table 2. gives the OET with a maximal set of marked entries[1].

	CUSTOMER	BRANCH	PRODUCT TYPE	PRODUCT TYPE IN BRANCH	PRODUCT	USAGE INTENTION	USAGE
create_customer	C						
modify_customer	M						
end_customer	E						
create_branch		C					
modify_branch		M					
end_branch		E					
create_product_type			C				
modify_product_type			M				
end_product_type			E				
allocate_product_type_to_branch				C			
modify_product_type_in_branch				M			
end_product_type_in_branch				E			
create_product					C		
modify_product					M		
end_product					E		
cr_usage_intention						C	
allocate_product					M	M	
confirm_availability						M	
cancel_usage_ intention						E	
start_usage						E	C
normal_return							M
abnormal_return							M
modify_conditions							M
invoice_usage							M
receive_payment							E
end_usage							E

Table 1. OET for the generic model with a minimal set of class/event involvements.

[1] The decision which entries to mark and which not to mark can be done by cross-checking the OET with the class diagram. A formal definition of the semantics of the OET and how to cross-check it with a class diagram are beyond the scope of this paper. The interested reader can find explanations, motivation and formal definitions in [10, 11].

	CUSTOMER	BRANCH	PRODUCT TYPE	PRODUCT TYPE IN BRANCH	PRODUCT	USAGE INTENTION	USAGE
create_customer	C						
modify_customer	M						
end_customer	E						
create_branch		C					
modify_branch		M					
end_branch		E					
create_product_type			C				
modify_product_type			M				
end_product_type			E				
allocate_product_type_to_branch		M	M	C			
modify_product_type_in_branch		M	M	M			
end_product_type_in_branch		M	M	E			
create_product		M	M	M	C		
modify_product		M	M	M	M		
end_product		M	M	M	E		
cr_usage_intention	M	M	M	M		C	
allocate_product	M	M	M	M	M	M	
confirm_availability	M	M	M	M		M	
cancel_usage_ intention	M	M	M	M		E	
start_usage	M	M	M	M	M	E	C
normal_return	M	M	M	M	M		M
abnormal_return	M	M	M	M	M		M
modify_conditions	M	M	M	M	M		M
invoice_usage	M	M	M	M	M		M
receive_payment	M	M	M	M	M		E
end_usage	M	M	M	M	M		E

Table 2. OET for the generic model with a maximal set of class/event involvements.

Lifecycles are written as regular expressions: a '+' denotes choice, a '.' denotes sequence and a '*' denotes iteration. The lifecycle expression should contain all events for which an entry has been marked in the corresponding column of the OET. In addition, the lifecycle expression should respect the type of the entries: events marked with a 'C' should appear as creating events, events marked with an 'M' should appear as modifying event types and events marked with an 'E' should terminate the life of the object. For example the lifecycle expression for the class USAGE is specified as follows:

USAGE = *start_usage* . (*modify_conditions*)* . (*normal _return* + *abnormal_return*). *invoice_usage* . (*receive_payment* + *end_usage*)

That is, after a usage has started, the conditions can be modified (e.g. postponing the return date) zero, once or more times. The product is then returned either in a normal state or in an abnormal state (e.g. crashed car). The usage is then invoiced and ends with the payment of the invoice or with the default *end_usage* event if the

invoice gets never paid. The lifecycle for USAGE_ INTENTION is:

USAGE_INTENTION = *create_usage_intenion* . *allocate_product* . *confirm* .
(*cancel_usage_intention* + *start_usage*)

When classes show some parallel behaviour the '||' symbol is used to denote parallel composition, such as in the lifecycle of product:

PRODUCT =
create_product .
[(*modify_product* + *allocate_product* + *invoice* + *receive_ payment* + *end_usage*)*
|| (*start_usage* . (*modify_conditions*)* . (*normal _return* + *abnormal_return*))*]
. *end_product*

That is, after a product has been created, its life is determined by two parallel threads. On the one hand there is the usage cycle and on the other hand there are a number of events that can occur randomly and independent from the usage cycle. The life of the product is terminated by the *end_product* event. Notice that constraints on event types such as *invoice* and *receive_payment* are already specified in the lifecycle of USAGE and need not be respecified in the lifecycle of PRODUCT. Lifecycles can also be specified by any kind of statechart, such as e.g. in UML.

b. Reuse of the generic model for the car rental company

For the car rental company, most event types can be reused. Some of them are renamed:
create_product_type becomes *create_car_model*
modify_product_type becomes *modify_car_model*
end_product_type becomes *end_car_model*
allocate_product_type_to_branch becomes *make_car_model_available_in_branch*
modify_product_type_in_branch becomes *modify_car_model_in_branch*
end_product_type_in_branch becomes *end_car_model_in_branch*
create_product becomes *buy_car*
modify_product becomes *modify_car_details*

end_product becomes *end_of_car*
cr_usage_intention becomes *reserve*
allocate_product becomes *allocate_car*
cancel_usage_ intention becomes *cancel _reservation*
start_usage becomes *rent*
modify_conditions becomes *change_return_date*
invoice_usage becomes *invoice*
end_usage becomes *end_rental*

Finally, the *abnormal_return* is split in two event types: *crash_car* and *total_loss.* The event type *repair* is added to allow to put a car back in circulation after a crash. The resulting OET is shown in Table 3.

The life cycles for RENTAL and RESERVATION become:

RENTAL = *rent* . (*change_return_date*)* . (*normal _return* + *crash_car* + *total_loss*). *invoice*. (*receive_payment* + *end_rental*)

RESERVATION = *reserve* . *allocate_car* . *confirm* . (*cancel_reservation* + *rent*)

The life cycle of CAR becomes more complex as we want to specify that after a total_loss a car can never be rented again and that after a crash, the car needs repairing.

CAR =
buy_car .
[(*modify_car_details* + *allocate_car* + *invoice* + *receive_ payment* + *end_rental*)*
|| (*rent* . (*change_return_date*)* . (*normal _return* + *crash_car.repair*))*
.(1 + (*rent* . (*change_return_date*)* .*total_loss)*]
. *end_product*

In this lifecycle the '1' stands for the empty event. The lifecycle thus specifies that after an arbitrary number of rent-cycles either nothing special happens or we have one final rent cycle that ends with the total-loss of the car.

	CUSTOMER	BRANCH	CAR MODEL	CAR MODEL IN BRANCH	CAR	RESER VATION	RENTAL
create_customer	C						
modify_customer	M						
end_customer	E						
create_branch		C					
modify_branch		M					
end_branch		E					
create_car_model			C				
modify_car_model			M				
end_car_model			E				
make_car_model_available_in_branch		M	M	C			
modify_car_model_in_branch		M	M	M			
end_car_model_in_branch		M	M	E			
buy_car		M	M	M	C		
modify_car_details		M	M	M	M		
end_of_car		M	M	M	E		
reserve	M	M	M	M		C	
allocate_car	M	M	M	M	M	M	
confirm_availability	M	M	M	M		M	
cancel_reservation	M	M	M	M		E	
rent	M	M	M	M	M	E	C
normal_return	M	M	M	M	M		M
crash_car	M	M	M	M	M		M
total_loss	M	M	M	M	M		M
repair	M	M	M	M	M		
change_return_date	M	M	M	M	M		M
invoice	M	M	M	M	M		M
receive_payment	M	M	M	M	M		E
end_rental	M	M	M	M	M		E

Table 3. OET for the Car Rental Company.

	MEMBER	LIBRARY	TITLE	COPY	RESERVATION	LOAN
register_member	C					
modify_member_details	M					
leave	E					
create_library		C				
modify_library_details		M				
end_library		E				
create_title			C			
modify_title			M			
end_title			E			
classify_copy		M	M	C		
modify_copy_details		M	M	M		
end_copy		M	M	E		
reserve	M	M	M		C	
cancel_reservation	M	M	M		E	
borrow	M	M	M	M	E	C
return	M	M	M	M		M
lose	M	M	M	M		M
renew	M	M	M	M		M
fine	M	M	M	M		M
receive_payment	M	M	M	M		E
end_loan	M	M	M	M		E

Table 4. OET for the Library

c. Reuse of the generic model for the library

For the library the event types are renamed as follows:

create_customer becomes *register_member*
modify_customer becomes *modify_member_details*
end_customer becomes *leave*
create_branch becomes *create_library*
modify_branch becomes *modify_ library_details*
end_branch becomes *end_library*
create_product_type becomes *create_title*
modify_product_type becomes *modify_title*
end_product_type becomes *end_title*
allocate_product_type_to_branch is dropped
modify_product_type_in_branch is dropped
end_product_type_in_branch is dropped
create_product becomes *classify_copy*
modify_product becomes *modify_copy_details*
end_product becomes *end_copy*
cr_usage_intention becomes *reserve*
allocate_product is dropped
confirm_availability is dropped
cancel_usage_ intention becomes *cancel*
start_usage becomes *borrow*
normal_return becomes *return*
abnormal_return becomes *lose*
modify_conditions becomes *renew*
invoice_usage becomes *fine*
receive_payment
end_usage becomes *end_loan*

The corresponding OET for the library is given in table 4.

The life cycles for LOAN and RESERVATION become:

LOAN = *borrow* . (*renew*)* . (*return* + *lose*). *fine* . (*receive_payment* + *end_rental*)

RESERVATION = *reserve* . (*cancel_reservation* + *borrow*)

In the life cycle of COPY we specify that after a copy has been lost it can never be borrowed again.

COPY =
classify_copy .
[(*modify_copy_details* +*fine* + *receive_ payment* + *end_loan*)*
|| (*borrow* . (*renew*)* . *return*)*.
(1 + (*borrow* . (*renew*)* .*lose*))]
. *end_copy*

6. Reuse Measurement Revisited

If both the class diagram and the OET are available, then more detailed reuse measurements can be taken. The reuse of dynamic business model aspects is to some extent taken into account by expressing the size of classes in terms of the OET entries that have been marked for them. It can be argued that the higher the number of event types with occurrences that affect a business class, the larger the size of that class as the class definition must somehow handle the response of objects to these event occurrences.

The definition of the reuse measures of section 4 can be reformulated such that they are sensitive to the size of a reused component and to the verbatim/non-verbatim character of reuse. Both aspects are desirable properties for reuse measures [2, 1]. A large verbatim reused component should contribute more to the reuse value than a small non-verbatim reused component.

The public reuse measure for the 'client' perspective is redefined as:

$$public\_reuse\_client(P) = \frac{size'(P_{ext})}{size(P_{ext}) + size(P_{new})},$$

where P_{new} and P_{ext} denote, as before, the newly developed and externally built parts of a software or specification artifact P, and *size'* refers to the 'size-contributing' features of P_{ext} that have effectively been reused from the 'server'. For the 'server' perspective, an analogous measure definition can be formulated.

For the extended car rental company, the degree of reuse is 100 % when evaluated from the perspective of the 'server' model. All marked entries in the OET in Table 2 have been reused. However, the OET in Table 3 contains 11 additional entries that are marked, resulting in a degree of reuse of 94/105 or 90 % from the perspective of the 'client' model. In particular, all classes affected by the occurrence of *repair* and *abnormal_return* (either *crash_car* or *total_loss*) events are reused non-verbatim. The reuse values for the extended library are respectively 100 % for the 'client' perspective and 62/94 or 66 % for the 'server' perspective. This latter value can be compared to the more coarse-grained measurement of 5/6 or 86 % degree of reuse of classes (cf. section 4).

7. Conclusions

Reuse of Domain models

Reuse of business models is certainly possible. But it remains true that generic business models must be carefully analysed and adapted where necessary to fit the particular business.

The generic model presented in this paper, typically represents any kind of renting business where resources or products are made available to customers. In this type of business, product as well as product types are considered relevant business classes. However, in a food store for example, we will only keep track of product types, but not of individual products. We sell bottles of skimmed milk, but will not keep track of selling transactions on the level of the individual bottle of milk. On the contrary, if we sell cars, it does make sense to keep a record of each individual car. This demonstrates that when reusing business models, one must very carefully judge

whether the model is applicable as such to the own situation or not. This critical judging of the reusability of business models can reveal opportunities and generate new ideas for organising ones own business

Business models as reusable Software requirements

Defining a business model is part of the requirements engineering step in the development of an information system: all business rules described in the business model have to be supported by the information system. Methods such as JSD [5], OO-SSADM [9], Catalysis [3] and MERODE [10, 11] even explicitly define domain modelling as a separate step in the development process. Jacobson [6] assumes the existence of a domain model that serves as a basis to identify entity objects. In object-oriented development, the domain object classes can be directly part of the class diagrams for the information system. As such business models are immediately reusable specifications.

The availability of a business model is also important when considering reusing existing software, such as for example, ERP packages. A possible evaluation criteria for such packages is the comparison of the own business model with the underlying domain model of the software that one wishes to reuse. The more the two models match, the easier reuse will be. Each discrepancy between the two models implies either a required adaptation of the software or an adaptation of the own business. The cost of such changes must be carefully evaluated before buying software. The measurements that we have presented in this paper are a good tool to quantify the necessary changes and estimate the related cost.

References

1. J.M. Bieman and S. Karunanithi, 'Measurement of Language-Supported Reuse in Object-Oriented and Object-Based Software', J. Systems and Software, vol. 30, no. 3, Sep. 1995, pp. 271-293.
2. P. Devanbu, S. Karstu, W. Melo, and W. Thomas, 'Analytical and Empirical Evaluation of Software Reuse Metrics', in: Proc. 18th Int'l Conf. Software Eng., Berlin, Mar. 1996, 19 pp.
3. D'Souza, Desmond F, Wills, Alan Cameron, Objects, components, and

frameworks with UML: the catalysis approach, Addison-Wesley Reading (Mass.), 1998

4. Hitz M., 'Measuring Reuse Attributes in Object-Oriented Systems', in : Proc. 1995 Int'l Conf. Object Oriented Information Systems (OOIS'95), Dublin, Dec. 1995, 8 pp.
5. Jackson, M. A., *System Development*, Prentice Hall Englewood Cliffs (N.J.), 1983, 418 pp.
6. Jacobson Ivar et al., *Object-Oriented Software Engineering, A use Case Driven Approach*, Addison-Wesley, 1992
7. Nellborn Christer, Business and Systems Development: Opportunities for an Integrated Way-of-Working, in Nilsson Anders G., Tolis Christofer, Nellborn Christer (eds.), *Perspectives on Business Modelling: understanding and Changing Organisations*, Springer Verlag Berlin, 1999
8. J. Poulin, J. Caruso, and D. Hancock, 'The business case for software reuse', IBM Systems J., vol. 32, no. 4, 1993, pp. 567-594.
9. Robinson Keith, Berrisford Graham, *Object-oriented SSADM*, Prentice Hall International (UK), 1994
10. Snoeck Monique, Dedene Guido, Existence Dependency: the key to semantic integrity between structural and behavioural aspects of object types, IEEE Transactions on Software Engineering, Vol. 24, No. 4, April 1998, pp. 233-251.
11. Snoeck Monique, Dedene Guido, Verhelst Maurice, Depuydt Anne-Marie, *Object-oriented Enterprise Modelling with MERODE*, University Press Leuven, 1999
12. Whitmire S.A., *Object Oriented Design Measurement*, Wiley Computer Publishing, New York, 1997, 452 pp.